Binary Search Tree

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

//Represent a node of binary tree

**struct** node{

**int** data;

**struct** node \*left;

**struct** node \*right;

};

//Represent the root of binary tree

**struct** node \*root = NULL;

//createNode() will create a new node

**struct** node\* createNode(**int** data){

    //Create a new node

**struct** node \*newNode = (**struct** node\*)malloc(**sizeof**(**struct** node));

    //Assign data to newNode, set left and right child to NULL

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

**return** newNode;

}

//Represent a queue

**struct** queue

{

**int** front, rear, size;

**struct** node\* \*arr;

};

//createQueue() will create a queue

**struct** queue\* createQueue()

{

**struct** queue\* newQueue = (**struct** queue\*) malloc(**sizeof**( **struct** queue ));

    newQueue->front = -1;

    newQueue->rear = 0;

    newQueue->size = 0;

    newQueue->arr = (**struct** node\*\*) malloc(100 \* **sizeof**( **struct** node\* ));

**return** newQueue;

}

//Adds a node to queue

**void** enqueue(**struct** queue\* queue, **struct** node \*temp){

    queue->arr[queue->rear++] = temp;

    queue->size++;

}

//Deletes a node from queue

**struct** node \*dequeue(**struct** queue\* queue){

    queue->size--;

**return** queue->arr[++queue->front];

}

//insertNode() will add new node to the binary tree

**void** insertNode(**int** data) {

    //Create a new node

**struct** node \*newNode = createNode(data);

    //Check whether tree is empty

**if**(root == NULL){

        root = newNode;

**return**;

    }

**else** {

        //Queue will be used to keep track of nodes of tree level-wise

**struct** queue\* queue = createQueue();

       //Add root to the queue

        enqueue(queue, root);

**while**(**true**) {

**struct** node \*node = dequeue(queue);

            //If node has both left and right child, add both the child to queue

**if**(node->left != NULL && node->right != NULL) {

                enqueue(queue, node->left);

                enqueue(queue, node->right);

           }

**else** {

               //If node has no left child, make newNode as left child

**if**(node->left == NULL) {

                    node->left = newNode;

                    enqueue(queue, node->left);

               }

               //If node has left child but no right child, make newNode as right child

**else** {

                    node->right = newNode;

                  enqueue(queue, node->right);

                }

**break**;

            }

        }

    }

}

//inorder() will perform inorder traversal on binary search tree

**void** inorderTraversal(**struct** node \*node) {

    //Check whether tree is empty

**if**(root == NULL){

        printf("Tree is empty\n");

**return**;

    }

**else** {

**if**(node->left != NULL)

            inorderTraversal(node->left);

       printf("%d ", node->data);

**if**(node->right != NULL)

            inorderTraversal(node->right);

        }

    }

**int** main(){

    //Add nodes to the binary tree

    insertNode(1);

    //1 will become root node of the tree

    printf("Binary tree after insertion: \n");

    //Binary after inserting nodes

    inorderTraversal(root);

    insertNode(2);

    insertNode(3);

    //2 will become left child and 3 will become right child of root node 1

    printf("\n Binary tree after insertion: \n");

    //Binary after inserting nodes

    inorderTraversal(root);

    insertNode(4);

    insertNode(5);

    //4 will become left child and 5 will become right child of node 2

    printf("\nBinary tree after insertion: \n");

    //Binary after inserting nodes

    inorderTraversal(root);

    insertNode(6);

    insertNode(7);

    //6 will become left child and 7 will become right child of node 3

    printf("\nBinary tree after insertion: \n");

    //Binary after inserting nodes

    inorderTraversal(root);

**return** 0;

}

Binary tree after insertion

1

Binary tree after insertion

2 1 3

Binary tree after insertion

4 2 5 1 3

Binary tree after insertion

4 2 5 1 6 3 7

Implement Graph Traversal techniques:) Depth First Search b) Breadth First Search

/\* Breadth First Search

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 5

struct Vertex {

char label;

bool visited;

};

//queue variables

int queue[MAX];

int rear = -1;

int front = 0;

int queueItemCount = 0;

//graph variables

//array of vertices

struct Vertex\* lstVertices[MAX];

//adjacency matrix

int adjMatrix[MAX][MAX];

//vertex count

int vertexCount = 0;

//queue functions

void insert(int data) {

queue[++rear] = data;

queueItemCount++;

}

int removeData() {

queueItemCount--;

return queue[front++];

}

bool isQueueEmpty() {

return queueItemCount == 0;

}

//graph functions

//add vertex to the vertex list

void addVertex(char label) {

struct Vertex\* vertex = (struct Vertex\*) malloc(sizeof(struct Vertex));

vertex->label = label;

vertex->visited = false;

lstVertices[vertexCount++] = vertex;

}

//add edge to edge array

void addEdge(int start,int end) {

adjMatrix[start][end] = 1;

adjMatrix[end][start] = 1;

}

//display the vertex

void displayVertex(int vertexIndex) {

printf("%c ",lstVertices[vertexIndex]->label);

}

//get the adjacent unvisited vertex

int getAdjUnvisitedVertex(int vertexIndex) {

int i;

for(i = 0; i<vertexCount; i++) {

if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)

return i;

}

return -1;

}

void breadthFirstSearch() {

int i;

//mark first node as visited

lstVertices[0]->visited = true;

//display the vertex

displayVertex(0);

//insert vertex index in queue

insert(0);

int unvisitedVertex;

while(!isQueueEmpty()) {

//get the unvisited vertex of vertex which is at front of the queue

int tempVertex = removeData();

//no adjacent vertex found

while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {

lstVertices[unvisitedVertex]->visited = true;

displayVertex(unvisitedVertex);

insert(unvisitedVertex);

}

}

//queue is empty, search is complete, reset the visited flag

for(i = 0;i<vertexCount;i++) {

lstVertices[i]->visited = false;

}

}

int main() {

int i, j;

for(i = 0; i<MAX; i++) // set adjacency {

for(j = 0; j<MAX; j++) // matrix to 0

adjMatrix[i][j] = 0;

}

addVertex('S'); // 0

addVertex('A'); // 1

addVertex('B'); // 2

addVertex('C'); // 3

addVertex('D'); // 4

addEdge(0, 1); // S - A

addEdge(0, 2); // S - B

addEdge(0, 3); // S - C

addEdge(1, 4); // A - D

addEdge(2, 4); // B - D

addEdge(3, 4); // C - D

printf("\nBreadth First Search: ");

breadthFirstSearch();

return 0;

}